SYSTEM AND METHOD OF CONTROLLING SOFTWARE DECOMPRESSION THROUGH EXCEPTIONS

Inventor: Christopher R. Risucci

BACKGROUND OF THE INVENTION

Field of the Invention

[0001] The present invention relates generally to the field of microprocessors and more particularly to the use of modified microprocessor instructions to transform stored processor code.

Background Art

[0002] Single chip computers are used in a wide range of applications where their small size and processing power are an advantage over conventional systems. When designing a single chip computer, or modifying an existing system for a new application, the system designer must balance the space available on the silicon chip against the space required by the components necessary to provide the desired functionality. Adding components to the computer system such as additional memory or digital to analog converters will add functionality but also will take up valuable chip space. If the chip size is fixed and all available space is already being used, it will not be possible to add functionality without removing a component of equivalent size. Reducing the number of components reduces the required chip size but the number of components cannot be reduced below the minimum necessary for a functional computer.

[0003] One component that must be included in single chip computers is memory. The amount of memory required is a function of the complexity and execution speed desired for planned computer software applications. Larger and more complex software programs usually require more memory to properly execute, and more memory usually increases program execution speed.

[0004] One way to reduce the memory required in a single chip computer is to store the computer's software code in a compressed form. Compressing the code allows it to be stored in a smaller amount of memory and thereby reduces the computer system memory requirements. There are several available algorithms suitable for compressing computer code. However, since the processor cannot directly execute code stored in compressed form, a method of decompressing the code prior to its execution by the processor is needed. Existing techniques decompress large amounts of code such as a subroutine or large blocks of a main program before executing it. These techniques require the code decompression to occur either in the boot process or by some means external to the system processor. When the system processor is not used for decompression, additional hardware is necessary to accomplish the decompression. This makes it difficult and expensive to incorporate software compression in existing systems.

[0005] A further disadvantage to these techniques is the additional memory required to store the larger volume of decompressed code while it is waiting to be executed by the processor. This disadvantage reduces the memory savings originally achieved by compressing the code.

[0006] What is needed is a simple and effective method of manipulating and executing compressed computer code without the disadvantages discussed above.

BRIEF SUMMARY OF THE INVENTION

[0007] In a central processing unit, a method for transforming data into an instruction for execution by the central processing unit. The transformation is triggered by first receiving a misaligned instruction address, generating a hardware exception and then, in response to the exception, executing an exception handling routine that transforms the data into an instruction for the central processing unit.

[0008] An advantage of this invention is its use of existing exception handling hardware as a trigger for the data transformation. This allows the invention to be used in any system with exception handling hardware.

[0009] Another advantage of this method is that hardware data transformation triggers, such as special interrupts, are not necessary. The present invention gives control of the data transformation trigger to the software programmer vice the hardware designer. This improves the portability of the invention between different hardware platforms.

[0010] Additionally, this invention provides a flexible software tool for optimizing the quantity of data or code decompressed to the amount of memory available to store that data or code. Precise control of the code and data decompression enables the programmer to ensure his program will execute in the available memory and while taking advantage of the storage space reduction allowed by compression.

[0011] The foregoing and other features and advantages of the invention will be apparent from the following, more particular description of a preferred embodiment of the invention, as illustrated in the accompanying drawings

BRIEF DESCRIPTION OF THE DRAWINGS/FIGURES

[0012] FIG. 1 illustrates a processor core embodiment of the invention.

[0013] FIG. 2 illustrates a method for using an exception handling routine to transform data.

[0014] FIG. 3 illustrates additional details of receiving a misaligned instruction, step 204 of FIG. 2.

[0015] FIG. 4 illustrates details of an exception handling transform, step 208 of FIG. 2.

[0016] FIG. 5 illustrates details of transforming data into an instruction, step 404 of FIG. 4.

[0017] FIG. 6 illustrates a computer system embodiment of the invention.

DETAILED DESCRIPTION OF THE INVENTION

[0018] The preferred embodiment of the invention is now described with reference to the figures where like reference numbers illustrate like elements. Furthermore, the left most digit of each reference number corresponds to the figure in which the reference number is first used. While specific methods and configurations are discussed, it should be understood that this is done for illustration purposes only. A person skilled in the art will recognize that other configurations and procedures may be used without departing from the spirit and scope of the invention.

[0019] Referring to Figure 1, a block diagram of a processor core 100 is shown. Core 100 provides an example hardware environment for implementing an embodiment of the invention. A person skilled in the relevant art will recognize that the invention is not limited to application in this example environment. In fact, after reading the following description, it will become apparent to a person skilled in the relevant art how to implement the invention in alternative environments.

[0020] Core 100 includes a program counter 112 coupled to a fetch unit 110, a decode unit 106, an execution unit 108, exception logic 104 and memory 114. When an exception condition does not exist, core 100 operates as follows: fetch unit 110 retrieves data from an address in memory 114 specified by program counter 112. Decode unit 106 decodes the data into an instruction and sends it to execution unit 108 where it is executed.

[0021] Under certain conditions, such as a software error, program counter 112 will provide an address to fetch unit 110 which is not allowed or does not exist. When this occurs an exception is generated by decode logic 106 and the address that was used in the attempted fetch is sent to exception handling logic 104. Exception handling logic 104 performs the operations necessary to provide the

address of a valid instruction to program counter 112 so that fetch unit 110 can retrieve an instruction and continue program execution.

[0022] The present invention uses exception logic 104, and a set of instructions known as an exception handling routine, to perform intentional, rather than error corrective, actions in response to software commands. In a preferred embodiment, processor instructions are stored in compressed form in memory 114. Before these instruction can be decoded by decode unit 106 they must be decompressed. The programmer responsible for implementing his program on a particular hardware suite determines the locations where compressed data will be stored in memory 114. He then configures software instructions to generate a misaligned address whenever processor instructions, stored in compressed form, are required for execution. The misaligned address is sent from program counter 112 to fetch unit 110 where it causes an exception error. The exception error causes core 100 to suspend its previous operation, and send the misaligned address to exception logic 104.

[0023] Exception logic104 sets up core 100 to execute a set of processor instructions stored in memory 114. These instructions constitute the exception handling routine which functions to processes the misaligned address and cause data to be transformed from a stored form into an executable instruction. The executable instruction is then stored in memory for use after the exception routine is complete. In an embodiment, the memory address containing the data to be transformed is offset a known amount from the misaligned address provided by the programmer. The programmer sets up the exception handling routine to add this offset to the misaligned address and retrieve the data stored at the offset location. Another embodiment of the invention uses a misaligned instruction address as the data to be transformed. A further embodiment uses the misaligned address and a programmer generated lookup table to generate the memory address containing the compressed data.

[0024] After retrieving the stored data, the exception handling routine applies a transformative algorithm to the data. In one embodiment the transformation is a

decompression algorithm. In other embodiments of the invention the programmer can select a number of transforming algorithms for implementation in the exception handling routine. A partial list of these algorithms are: decrypting an encrypted instruction, decoding a macro instruction, transforming a non-native instruction into a processor executable instruction and executing a random number of processor instructions. Based on the foregoing description, it will be apparent to a person skilled in the relevant art that other transformations can be implemented in accordance with the invention.

[0025] The retrieval, transformation and storage of processor instructions continue until the desired data has been transformed and stored in memory for subsequent execution by the processor.

[0026] When the data transformation is complete, the exception handling routine provides the address of the first transformed instruction to program counter 112, and returns core 100 to the pre-exception condition. Program counter 112 then provides fetch unit 110 with the address of the first transformed instruction which is decoded by decode unit 106 and executed by execution unit 108. Program execution continues as directed by the software program being executed.

[0027] FIG. 2 illustrates a method of triggering data transformation according to the present invention. In step 204, a misaligned instruction address is received by fetch unit 110. As indicated by step 206, the misaligned instruction address causes an exception. Exception logic 104 then executes an exception handling routine to transform the data in a step 208.

[0028] FIG. 3 describes step 204 in further detail. In step 304, a "jump to" instruction is executed causing fetch unit 110 to attempt to retrieve the next processor instruction from a specified "jump to" address. In a preferred embodiment of this invention the software programmer designates the jump address as an "odd" address, defined as an address with a least significant bit value of one. A core 100 with a 16 bit (4 bytes) address bus stores instructions at even byte boundaries in memory. In an embodiment, a misaligned instruction address is defined as an "odd" address. Therefore, fetch unit 110 receives a

misaligned instruction address in a step 204. It will be apparent to a person skilled in the relevant art that other programming techniques can be used to cause the processor to receive a misaligned instruction address.

[0029] FIG. 4 describes step 208 in further detail. In a step 404, the misaligned data is transformed into at least one instruction. Step 404 continues until all data specified by the programmer is transformed. The first transformed instruction is stored at a first address in memory 114 in a step 406. The first address in memory 114, where the transformed instructions are stored, is then loaded into program counter 112 in a step 408. A return from exception is executed, in a step 410, to return core 100 to its pre-exception mode of operation. Fetch unit 110 then retrieves a transformed instruction (starting with the address indicated by program counter 112) for execution as indicated by step 412.

[0030] Step 404 is described in further detail with reference to FIG. 5. In a step 502 an offset value is added to the misaligned instruction address. Data stored at the address defined by the addition of the offset value and the misaligned instruction address is retrieved in a step 503. A transformative algorithm is then applied to the retrieved data in a step 504. The result of the transformation is designated a processor instruction in a step 506.

Example Exception Handling Routine

[0031] When an exception is triggered the core 100 automatically executes a set of instructions known as an exception handling routine. An example of the software code embodying an exception handling routine is provided below. Additional information on programming exception handling routines is found in, Dominic Sweetman, See MIPS Run (1999), which is incorporated herein by reference in its entirety. Although this reference is for a particular hardware set, persons of skill in the art will understand how to implement the present invention on other hardware platforms.

[0032] The example routine is written in a C language format. This code is for illustrating the basic operation of an exception handling routine and cannot be compiled.

```
/* The hardware exception logic transfers program execution to a
preprogrammed address where a software exception handler is stored. For this
example the exception handler is called Execption_handler. The bad address is
data that caused a hardware exception condition in the core 100.
*/
Execption_handler()
{
        /* Check to see if this is an exception for a bad address.
        */
        if (conduct hardware dependent check for bad address);
        {
                /*
                The hardware exception logic passes the bad address to the
                software exception handler (the detailed method is hardware
                dependent). The software exception handler uses the bad address
                to determine if the exception is a decode trigger or an actual bad
                address. To do this the hardware may have a list of valid address
                that are acceptable or a range or address that are acceptable or just
                check to see if it is a odd address (lowest order address bit set).
                */
                if (valid decode address);
                {
                        /*
                        If the software exception handler determines the cause of
                        the exception is a purposeful decode trigger, it may use
                        the bad address to determine where the encoded data is
                        stored and decode that encoded data into a location in
```

memory from which it can later be executed. If a transformation of the data is desired, a function, decode, is call to perform that function. The decode function will decide when to stop decoding. For example if this were a program instruction, being decoded, a good stopping point would be the next branch instruction. The decode function will return a value, decode_return. This value could represent an address of a decoded section of code that the core 100 can use to continue execution after it leaves the decode function or as an address where the newly decoded data is stored.

*/

decode_return = decode(bad_address)

/*

What the software exception handler does depends on what was decoded. For this example, if the decoder decoded compressed instructions and the decode_return value was the address of the first instruction of a block of instructions that were decoded. The decode function will need to "fix" the address so the core 100 will know where to begin execution after the exception handling function finishes. In many core 100 designs there is a register in the core 100 called a program counter112 which is the address the core 100 uses to fetch the current instruction or the next instruction that will be executed. The program counter 112 may still contain the bad address that caused the exception. This address needs to be changed to the address of the decompressed code so the core 100 will begin executing of the decompressed code.

*/

```
                fix_program_counter(decode_return);
            }
            else
            {
                /*
                If this was not a valid decode address or exception then
                the exception function will continue here with logic to
                process the exception.
                /*
            }
        }
        else
        {
            /*
            Check for other exceptions
            */
        }
        /*
        At this point the exception function will do any addition house keeping
        that is need for the particular core 100 architecture and return.
        */
        return;
}
```


[0033] The invention can be implemented in a computer system capable of carrying out the functionality described herein. An example of a computer system 600 is shown in FIG. 6. Various software embodiments are described in terms of this example computer system. After reading this description, it will become apparent to a person skilled in the relevant art how to implement the invention using other computer systems and/or computer architectures. Computer system

600 includes one or more processors, such as processor 605. Processor 605 is connected to a communication bus 606.

[0034] Computer system 600 also includes exception handling hardware 604, a main memory 614, preferably random access memory (RAM), and may also include a secondary memory 610. The secondary memory 610 may include, for example, a hard disk drive 612 and/or a removable storage drive 613, representing a floppy disk drive, a magnetic tape drive, an optical disk drive, etc. The removable storage drive 613 reads from and/or writes to a removable storage unit 618 in a well-known manner. Removable storage unit 618, represents a floppy disk, magnetic tape, optical disk, etc. which is read by and written to by removable storage drive 613. As will be appreciated, the removable storage unit 618 includes a computer usable storage medium having stored therein computer software and/or data.

[0035] Secondary memory 610 may include similar means for allowing computer programs or other instructions to be loaded into computer system 600. Such means may include, for example, a removable storage unit 622 and an interface 620. Examples of such may include a program cartridge and cartridge interface (such as that found in video game devices), a removable memory chip (such as an EPROM, or PROM) and associated socket, and other removable storage units 622 and interfaces 620 which allow software and data to be transferred from the removable storage unit 622 to computer system 600.

[0036] Computer system 600 may also include a communications interface 624. Communications interface 624 allows software and data to be transferred between computer system 600 and external devices. Examples of communications interface 624 may include a modem, a network interface (such as an Ethernet card), a communications port, a PCMCIA slot and card, etc. Software and data transferred via communications interface 624 are in the form of signals 628 which may be electronic, electromagnetic, optical or other signals capable of being received by communications interface 624. These signals 628 are provided to communications interface 624 via a communications path (i.e., channel) 626.

This channel 626 carries signals 628 and may be implemented using wire or cable, fiber optics, a phone line, a cellular phone link, an RF link, and other communications channels.

[0037] Computer programs (also called computer control logic) are stored in main memory 614 and/or secondary memory 610. Computer programs may also be received via communications interface 624. Such computer programs, when executed, enable the computer system 600 to perform the features of the present invention as discussed herein. In particular, the computer programs, when executed, enable the processor 605 to perform the features of the present invention. Accordingly, such computer programs represent controllers of the computer system 600.

[0038] In some embodiments the invention is implemented using software, the software may be stored as computer program product and loaded into computer system 600 using removable storage drive 614, hard drive 612 or communications interface 624. The control logic (software), when executed by the processor 605, causes the processor 605 to perform the functions of the invention as described herein.

[0039] In other embodiments, the invention is implemented primarily in hardware using, for example, hardware components such as application specific integrated circuits (ASICs). Implementation of the hardware state machine so as to perform the functions described herein will be apparent to persons skilled in the relevant art(s).

[0040] In further embodiments the invention is implemented using a combination of hardware and software.

[0041] The invention can be implemented in software that describes hardware and is disposed, for example, in a computer usable (i.e., readable) medium configured to store the software (i.e., a computer readable program code). The program code causes the enablement of the functions or fabrication (or both) of the systems and techniques described above. This may be accomplished, for example, through the use of general programming language (e.g., C, C++),

hardware description language (HDL) including Verilog HDL, VHDL and so on, or other available programming and/or circuit (i.e., schematic) capture tools. The program code may be disposed in any known computer medium including semiconductor, magnetic disk, optical disc (e.g., CD-ROM, DVD-ROM) and as a computer data signal embodied in a computer usable (e.g., readable) transmission medium (e.g., carrier wave or any other medium including digital, optical, or analog-based medium). As such, the code can be transmitted over communication networks including the Internet and intranets.

[0042] It is understood that the functions accomplished and/or structure provided by the systems and techniques described above can be represented in a core (e.g., a microprocessor core) that is embodied in program code and may be transformed to hardware as part of the production of integrated circuits.

[0043] While a preferred and alternate embodiments of the present invention have been described above, it should be understood that it is presented by way of example, and not limitation. It will be apparent to persons skilled in the relevant art that various changes in form and detail can be made therein without departing from the spirit and scope of the invention. Thus the present invention should not be limited by any of the above-described exemplary embodiments, but should be defined only in accordance with the following claims and their equivalents.